The TDL Advantage

C. Farcas, M. Holzmann, H. Pletzer, G. Stieglbauer

Department of Computer Science
University of Salzburg
Austria

Technical Report T002
April 2004
http://cs.uni-salzburg.at/pubs/reports/T002.pdf

The TDL Advantage

C. Farcas, M. Holzmann, H. Pletzer, G. Stieglbauer

University of Salzburg
Institute for Computer Science

April 19th, 2004

Abstract

This report compares industrial state-of-the-art products (TTTech, DaVinci and dSPACE) for development of dependable real-time applications with the methodology introduced by Giotto and refined by the Timing Definition Language (TDL). Giotto/TDL aims at a paradigm shift in the development of dependable real-time control systems by focusing on the application rather than on the platform. It represents a high level language for describing the timing behavior of real-time applications and thereby abstracts from the execution platform (we call that property software standardization), provides determinism, allows modularization of applications and the straight-forward distribution of components. A simplified brake-by-wire example is used to evaluate the various approaches and highlight their advantages and disadvantages with respect to determinism, composability and software standardization.

Table of Contents

1 Executive Summary

Although it is a difficult task to compare commercial products to a fairly new academic approach, the comparison indicates that the available products fail to achieve the goals aimed at by Giotto and TDL, in particular, regarding determinism, composability, and software standardization. These software properties are explained in Section 2 of the report. TDL allows a platform-independent specification of the timing and commmunication behavior of a control system.

(1) This implies **significantly improved reusability and portability** of TDL applications compared to state-of-the-art approaches considered in this study.
(2) In addition to that, **TDL has introduced the module construct as basis for component-based, modular development of embedded control systems**. None of the other systems and tools offers a comparable abstraction.
(3) Finally, **TDL components can easily be distributed** without taking the distributed platform into account from the beginning. In other words, **TDL abstracts from the distributed platform.** The platform details are specified separately from the TDL program and later in the development cycle.

2 Introduction

Typical development of control systems software focusing on a specific platform faces significant challenges when maintenance and upgrades are needed. Changing the platform means redesigning and recoding large parts of a system. Timing and functional behaviors are tightly coupled and the system is running with lots of hand-coded tweaks for stability and safety.

A new approach for building control systems software relies on model-based development supported by mathematical models and automatic code generation. The Timing Definition Language (TDL) [1], a successor of Giotto [2], aims at a complete logical separation of functionality and timing behavior in hard real-time control systems for better maintenance, platform independence and inter-platform and time safety guarantee. This model based development approach is based on the FLET assumption that ensures predictability and determinism, time safety checking and zero jittering.

There are several commercial solutions available on the market that claim to address at least some of the goals of the TDL approach. Those solutions will be compared to the TDL model-based approach using the following requirements for dependable real-time software components as comparison base: determinism, composability, and software standardization.

For each of the compared tool chains we will present an overview, the required development steps, and a summary. At the end of the report we will summarize the overall comparison results. The development steps per tool chain will be presented in form of use cases for two typical examples, where the actor is always the programmer. Only the main flow of events resulting from the interaction between the programmer and the tool chain is presented. The resulting application will be analyzed per use case.

Aspects of development steps that have a relation with the TDL approach will be included in the description of the step and highlighted by using *italic* style.

Please note that the comparison involves commercial products with hundreds of person years invested with a rather new academic approach, which of course has not the same level of maturity, documentation, graphical user interfaces, etc. Therefore we focus more on the concepts behind those tools as on the graphical appearance and the details of using the tools. It was also not possible for us to install one of the tools in our lab (DaVinci), so we can only infer the properties and usage from the publicly available documentation for this tool chain.

2.1 Determinism

In general, determinism means that the behavior of the control system software is predictable. If a software component is called twice with the same input signals, it has to produce the same output signals (this applies to values and timing). As long as only the values of inputs and outputs are concerned, we speak of *value determinism*. In the case of real-time applications, however, we also have to consider the timing behavior of a component. If a component also preserves the timing behavior of its outputs, we speak of *time determinism*.

Example: If a control system receives the same inputs (sensor values) at the same time, it always has to react exactly in the same way.

Consequences:

- Testability: the behavior can be simulated and reproduced.
- Minimal jitter[1]: outputs are available close to ideal time.

2.2 Composability

The behavior of a software component has to be independent of the overall system load and configuration. Composability of a real-time system comprises two aspects:

- Functionality: the system is based on components (modules, packages, libraries, etc.) that can be reused or extended for additional purposes.
- Timing: the components that form the system may be composed independently without loosing their timing behavior.

Example: A new component can be added to a system without influencing the behavior of the original components.

Consequences:

- Extensibility of systems: new components for supplementary functionality can be added to existing systems.
- Reuse of components: existing components can be used to build/extend systems.

[1] the short-term variations of the significant instants of a digital signal from their ideal positions in time. (refer to Telecom Glossary [3])

2.3 Software Standardization

The behavior of a software component should be specified independent of its implementation.

Example: The hardware, operating system, or bus architecture can be changed without changing the behavior of the application components.

Consequences:

- Upgradeability of hardware: keep up with latest hardware achievements.
- Portability of software: migrate software from one platform to the other.
- Software components can be moved between ECUs.

3 Case Studies

As there are similar features in each tool-chain, the best way to compare them is to use a close to reality case study and analyze the developments steps and the quality of the resulting application.

3.1 Distributed Brake-By-Wire

The first case study focuses on a brake-by-wire system reduced for simplicity to the front part of the car. It is implemented as a software application running on two ECU's, each one connected to the corresponding wheel sensor and braking actuator, pedal sensors (two sensors for fault tolerance) and some output of the remote ECU (see Fig. 3.1). The communication channel (CAN, TTCAN or equivalent) is used to transport the values computed by the software running on each ECU to the other ECU to provide a consistent view of the system state. The transported values along with the sensor values are used to compute the next actuator output.

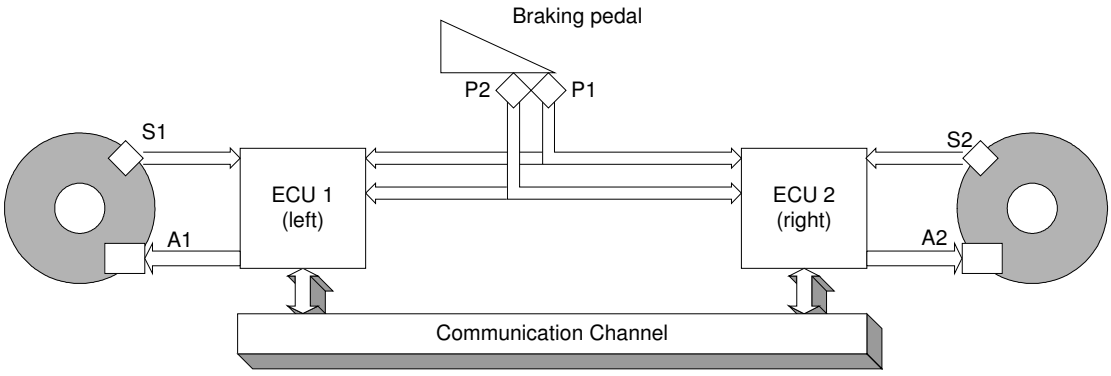

Figure 3.1 – Physical connection diagram.

The software application that implements this simplified brake by wire concept is implemented as two tasks, one on each ECU, that process the wheels and pedal sensor values and compute the output for the braking actuator. The period of the task is determined by the processing power of the ECU and the control law implemented by the two tasks (typically a couple of milliseconds).

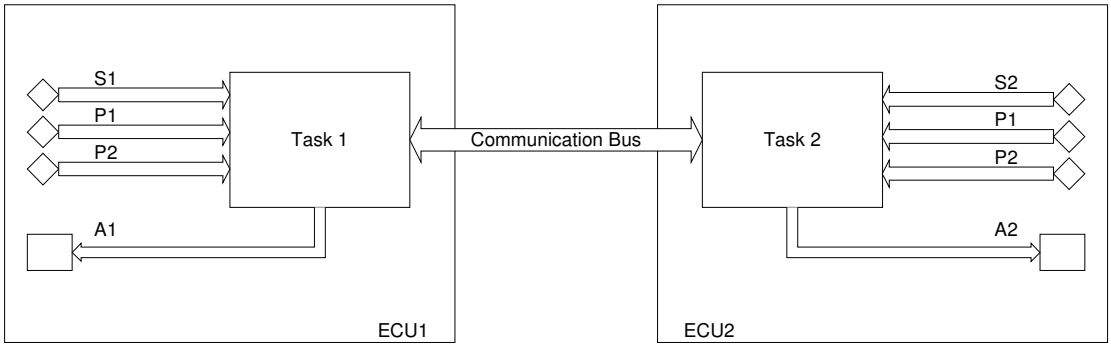

Figure 3.2: Software architecture diagram.

The whole system must be able to correctly read the sensors at precise time intervals and react with minimal jitter by setting the actuators to the computed position. As the functionality of both ECUs is identical (just the inputs/outputs are different) a component based design is desired.

3.2 ECU Consolidation

In the future faster CPUs will allow reducing the number of ECUs in a brake by wire system and will include all previous functionality. The tasks from each old ECU might require small functional changes to act like a real brake by wire system, but in this example, we presume that a simple duplication of the old functionality will suffice.

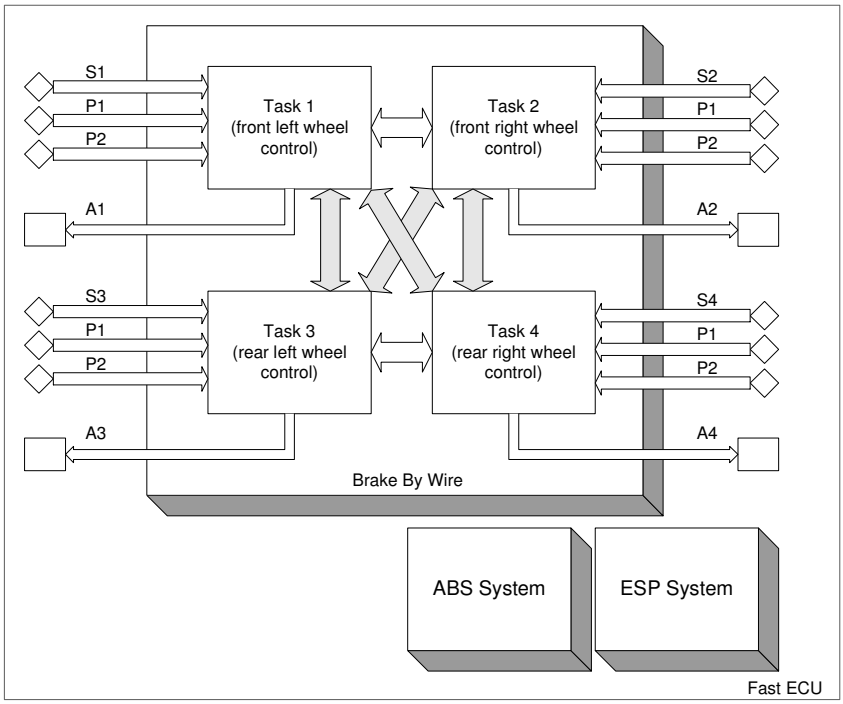

Figure 3.3 – Multiple applications on the same ECU.

Composing also the ABS and ESP functionality will further increase the requirements for dependable computing and time safety guarantees.

The second case study focuses on the steps required for replacing the previous two-ECU solution by a more powerful single-ECU node while maintaining the overall

system behavior in terms of functionality and timing in order to prevent redesigning the control laws. This kind of activity is also known as ECU consolidation.

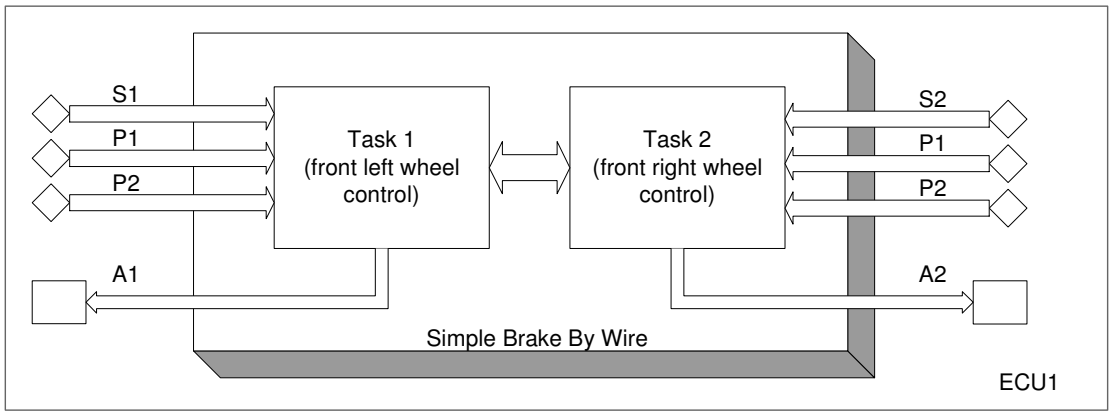

Figure 3.4 – Case study 2 - Software architecture diagram.

4 TTP Tools by TTTech

4.1 Overview

The time-triggered protocol (TTP) is a communication protocol for fault-tolerant distributed hard real-time systems. It provides time-triggered transmission of messages, distributed clock-synchronization and a membership service. The communication on the bus is done with static and periodic TDMA (time division multiple access) rounds.

TTTech provides a tool chain consisting of two main tools for application development for their hardware in order to realize their two-level design approach, which consists of cluster and node level design. The idea is that the system integrator knows all functions and communications, therefore also the bus-messages that are needed for them. After the first level is done, the outcome can be passed on to various sub-manufacturers, which design specific nodes of the cluster. The composability of the software running on the TTP nodes is guaranteed because the bus schedule is already generated in the development phase.

According to TTTech, the following distinct steps have to be taken in order to design a TTA application:

- Application design (including control algorithms)
- Communication and fault tolerance requirements
- TTP/C cluster schedule design
- Implementation
- Test/verification

TTPplan is the network level development tool for designing the network bus schedule (= cluster schedule). Every message that is sent from any node has to be specified and an automatic scheduler then generates a bus schedule matching the specifications. In addition, fault tolerance properties like replication, reintegration, and redundancy must be specified at this development stage. The outcome of TTPplan is a database containing the cluster network schedule and a MEDL (message descriptor list) for the TTP chip of each node (which contains the cluster schedule). The MEDL specifies

exactly when a node may send a certain message and when messages from the other nodes can be received.

TTPbuild requires a finished cluster database before designing a single node of the cluster. With this tool, one can specify every periodic task that should run on a node and the messages it consumes and produces. It generates a static schedule table with all user-defined tasks and with system task used for time synchronization and the fault-tolerant communication layer (FT-Com). The FT-Com layer is automatically generated C code for handling the reception and transmission of redundant or replicated messages. Another outcome of the tool is an OS configuration file for the TTTech operation system TTPos in which the schedule is specified. All the developer has to do to get a working binary image for the target platform is to provide the task functionality as C code.

4.1.1 Determinism

The TTP architecture does not use any runtime scheduling. A static, periodic schedule is generated automatically (can be "tuned" manually afterwards) by the TTP tools and is then carried out by the operating system TTPos. Therefore, value and time determinism is guaranteed at runtime. However if actuators are updated at the end of a task, jittering is non-deterministic.
With this concept, it is of course impossible to extend the system at runtime. In every application cycle the system behaves exactly the same way. However, at the cost of greatly reduced flexibility this approach delivers high reliability and minimal jitter.

4.1.2 Composability

According to the two-level design approach (see basic introduction on the tools), first a cluster schedule (for the whole network running the distributed application) has to be specified and generated. At this point all software components (or at least which values have to be communicated) of the whole system must be known in order to plan a suitable cluster schedule. If a software component that needs to transfer data on the bus is added, the cluster schedule has to be redesigned and consequently all nodes which depend on it. Typically adding a component means to redesign and recompile every node in the cluster, even if the system does not even execute any functionality of the added software component.

Composability is only possible if it is already known from the beginning what messages each component needs to transfer. However, once this is accomplished, the individual nodes and their components can be developed and tested independently and their composability is guaranteed on cluster-level. It is still not assured that individual components can be composed on one node, since due to e.g. lacking CPU time it might not be possible to find a valid task schedule.

4.1.3 Software Standardization

The flexibility here is very limited with the TTP tools because of the fact that the tools are tailored specifically to the use of the TTP protocol, the corresponding communication controller chips and the operating system TTPos that is part of the tool chain as well. The tools support a certain range of hardware, though, but still changing the type of communication controller might lead to a redesign of the cluster schedule and consequently the whole system.

There is an interface (TTPmatlink) to be able to use the tools in conjunction with Matlab/Simulink and so it is possible to reuse existing Simulink models to a certain degree. But again you have to use special TTP blocks to generate a suitable model for the TTP tools which of course will be incompatible with any other tool.

4.2 Use Case 1: Distributed Brake-By-Wire

4.2.1 Main Flow of Events

1) **System requirements analysis.** In this step, the overall system requirements have to be defined. This includes specifying the control laws and communication and fault-tolerance requirements. Actually this step is not accompanied by any part of the standard TTP tools but is vital for using them later. Tools like Matlab/Simulink can be used for this step. The TTP tool chain needs this analysis as a basis for all subsequent steps and therefore a later change in the control law timing requirements, for example, would mean to repeat and partly redesign all subsequent steps. A small error here can become very costly and therefore has to be avoided.

   This step has to be taken in a very similar way for TDL, but since in TDL timing is separated from functionality and other aspects such a change can be performed with much less complications.

2) **Cluster design (communication scheduling).** This step can be summarized by "who sends what at which time", so the communication pattern between nodes has to be defined explicitly. It has to be known what entity of functionality is executed on which node here. For the case study it has to be determined what values the two ECUs have to exchange and at what rate this exchange should occur. Only periodic (time-triggered) messages are supported by TTP. TTPplan is used to create messages that have to be transferred over the bus. For every message: the period, message type, the node that sends the message and fault-tolerance properties like redundant senders have to be given. In the case study we would create two messages (one sent from each node). In this step it is only relevant what goes on the bus, it is e.g. not about the exchange of messages between tasks inside a node. However, it must be known at this step what functionality each node provides and needs. Again, a careful analysis is advisable here, because all subsequent steps (especially the node design) rely on the outcome of this development stage.

   In TDL this step does not exist explicitly, but the communication requirements emerge out of the specification of the distribution of the TDL modules. By analyzing which modules need to exchange values between each other and whether they are on different ECUs or not a communication schedule is generated automatically.

3) **Node design (task scheduling).** On basis of the cluster schedule, a single node of the cluster has to be specified. It is not possible to design a node without finishing cluster design first. For every node (=ECU) the tasks that are supposed to run on it and what messages they use from the bus has to be defined. For scheduling purposes, the TTP tools need the WCET of each task. If no more restrictions (like period and phase of the task invocation) are set on the

scheduling of the task, the schedule will be automatically generated by the TTPbuild tool, which typically tries to minimize the interval between the reception of all relevant messages for the task and its invocation. The jitter is minimal due to the static schedule TTPbuild uses and a distributed time synchronization algorithm.

In TDL the period and WCET has to be specified for each task. The exact task invocation time is determined by the compiler and cannot be influenced, but because of the FLET property of TDL the exact invocation time is irrelevant. Sensors and actuators are read and set at precisely defined instances of time.

4) **Implementation of tasks**. The functionality code for all tasks has to be provided in C. The driver code for sensors and actuators might be included in the functionality task code or put in separate tasks, depending on how the system has been designed in the previous step. TTPbuild outputs C code containing TTPos configuration files that contain the task schedule and the FT-Com layer. These files are then compiled and linked together with the task code that is added by the developer to generate the binary image for each node.

When using TDL this step is similar but with the exception that drivers are not allowed to be included in the task code. The TDL constructs for specifying sensors and actuators have to be used. For every task and driver functionality code has to be provided.

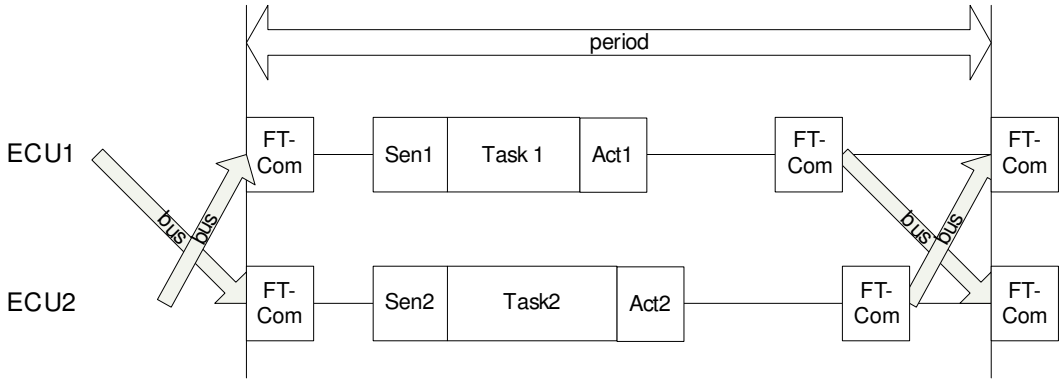

Figure 4.1 – Task invocation diagram for case study 1.

Figure 4.1 illustrates the tasks each ECU executes. The FT-Com layer tasks are automatically generated and scheduled by TTPbuild and handle the transmission of messages over the TTP bus. Sensors are read at the beginning of the computation tasks and actuators are set at the end. A way to design it with minimal jitter would be to introduce a separate task for setting the actuator and linking it via a message to the corresponding task. This requires more design effort and the overhead of additional messages.

4.2.2 Analysis of the Application Runtime Behavior

Determinism. Because of the static schedule and modeling the actuator update as a separate task, the system is value and time deterministic. This is reached at the cost of

very careful analysis in the early development steps, extra work for adding and scheduling messages from task outputs to a common actuator updater task and handling actuator updates in the common tasks. In practice when there are more tasks and actuators involved this may become impractical to develop and maintain.

4.3 Use Case 2: ECU Consolidation

One thing to note is that this case study is not fully applicable to the TTP architecture and tools because they are intended solely for distributed applications. Still the idea behind choosing this example, namely the evaluation of the ability to compose software components without loosing the value and time properties, can be analyzed. Furthermore the mentioned "faster hardware" must of course be supported be the TTP tools.

4.3.1 Main Flow of Events

1) **System requirements analysis.** The data already gathered for case study 1 can be reused, but additional analysis is required for the supplementary components. These components might interfere with the old ones and therefore they might have to be redesigned as well.

2) **Cluster design (communication scheduling).** This step is not applicable since the second case study is not a distributed application.

3) **Node design (task scheduling).** Identical behavior of the old and the new system can only be guaranteed if all tasks are executed at the exact same instant as before. This can be done to some degree by manually making the schedule identical, but this will be sometimes impossible (especially with distributed applications) and certainly will restrict node design. The schedule might be very inefficient and may prevent the scheduling of other tasks especially because the TTP tools do not support the preemption of tasks. It will be sufficient to take care that the sensors and actuators are executed at the exact same instant as before. Also the functional code for the sensors and actuators may have to be changed to adapt to the hardware changes.

4) **Implementation of tasks.** The already existing code from the previous case study can be reused and create identical behavior of the system if the timing was made right in the previous step. The compilation works like previously.

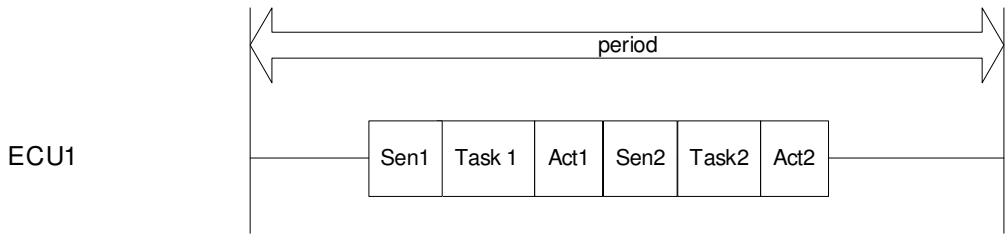

Figure 4.2 – Task invocation diagram for case study 2.

Figure 4.2 shows how the tasks are executed on the single node ECU1. When comparing to Figure 4.1 it can be seen that the time and value behavior is different. Task1 will behave approximately like before, but as a different CPU is used it is very likely that the actuators are set earlier due to the shorter execution time of the task. Task2 executes much later than before but gets the value from Task1 sooner. It can be expected that this timing differences have a considerable impact on the overall system behavior.

To achieve a solution with identical behavior than in the first case study, the sensors of both tasks have to be read before the actual computation of the task starts. Also the actuators have to be set with only minimal time between them. Furthermore it has to be assured that Task2 does not directly read the output of Task1 but only in the next period to get the same communication pattern as in the previous case study. So overall to create identical behavior the existing application has to be redesigned considerably.

4.3.2 Analysis of the Application Runtime Behavior

Time and value determinism. The schedule must be manually tuned to match the previous behavior, requiring significant effort. In practical applications with multiple tasks this is impossible. However when the schedule is available the system is both value and time deterministic (when actuator updates are handled in a separate task) and with minimal jitter.

Composability. When changing the application according to the second case study the programmer has two options: (1) either to begin from the start with the TTP tools and redesign and test all components again or (2) to try to model the task and cluster timing in a way so that it corresponds to the previous application. Both solutions require considerable effort and the latter might be impossible for distributed applications and requires the original application to be designed in a way such that it is easier to duplicate the timing for example by setting the actuators in a separate task.

In contrast, TDL is composable because of changing the hardware/software mapping (e.g. adding a second task to an ECU) does not affect the runtime behavior of a TDL controller because of the fact that the behavior of each task is defined independently of the platform in a former development step.

TDL is much more flexible here since a change of e.g. the period of a task does not require the developer to restart from the beginning again and still delivers value and time deterministic systems.

Software standardization. The TTP tools are tailored specifically to the supported TTP hardware and therefore the flexibility of the tools is very limited. From the first development step on the target platform must be known in detail and a change would imply a redesign of the whole system or at least a thorough check if all the analysis and design is still valid after the change. For complex models with lots of tasks this requires a large effort and might be impractical.

In contrast, if the TDL tool chain is used, so-called E-code is generated to handle the runtime behavior. E-code is platform independent and is interpreted by a virtual machine called E-machine. Only the E-machine has to be ported to a specific platform, which can be done very easily as a result of the small instruction set of E-code.

4.4 Summary

The TTP case study shows that it is to some extent and only under certain conditions it is possible to reuse already developed and tested software components with the TTP tools. Nevertheless, doing so requires either effort in designing the component in a way such that a later reuse is made easier, or a lot of effort to recreate identical time and value behavior in the new system. The tools do not support one of these procedures, which mean the user has to take care of this all and makes the development more complicated and prone to errors.

TDL supports compositionality natively and so the user does not have to care about it. In addition, the two-level design approach might no longer be necessary. TDL modules can be developed independently and finally be combined. Opposed to the TTTech tool chain, the cluster schedule does not have to be specified at the beginning.

TDL provides more flexibility regarding the usage of hardware (CPU and communication bus) and the addition of an extra software component would no longer mean a redesign of the whole system.

5 DaVinci Tool Suite by Vector Informatik

5.1 Overview

DaVinci, provided by Vector Informatik, supports the control engineer to develop applications for distributed platforms. It provides two tools which separate the design of functional components (DaVinci DEV) from the integration of those components to a specific, possibly distributed platform (DaVinci SAR).

In DaVinci, the functional components are called Software components, although they are based on the principles of functions and methods used by common imperative programming languages. A Software component is a ‘black box’, which provides some signal interfaces. Inside the box, the Software component consists either of further Software components (sub-functions, sub-methods) or of a behavior description (an implementation). The behavior description is expressed either by C-code or by a model (e.g. a finite-state-machine), which is designed by using a common modeling tool (e.g. MATLAB/Simulink).

In the first step, the target hardware platform and its topology has to be evaluated. During this evaluation, information about available sensors and actuators, the network structure (with ECUs as nodes) and the bus type (with the corresponding bus protocols) has to be collected. After that, the previously defined Software components are mapped to the evaluated hardware platform (e.g. which sensor is connected to which component, which components run on a specific ECU).

If modeling tools are used to describe the functional behavior of a Software component, the functional code is usually generated automatically by integrated code generator tools that produce C-code (or similar) out of the model. The code generators are integrated into DaVinci such that the generated functionality code fits into the rest of the application. The rest of the application consists mainly of sensor, actuator and bus drivers. These drivers are automatically generated from the information provided by the

DaVinci SAR tool and are responsible for bus communication and accessing sensors and actuators.

Testing of Software components is possible in several ways: If a modeling tool is used for defining the Software components, the simulation capabilities of this tool are used to verify the behavior of the Software components. Therefore, this testing strategy does not belong directly to DaVinci. The generated code, however, is tested by using a PC emulation of the OSEK operating system. The third testing possibility that is provided by DaVinci is to test the behavior of the Software components, which are executed within a PC environment, with a real hardware bus.

5.1.1 Determinism

Kirsch [4] distinguishes between three programming models in the area of real time programming:

- The *scheduled model*, which is based on traditional programming strategies such as optimized scheduling algorithms where it is presumed that the computation is fast enough to react in time to an external event.
- The *synchronous model* is based on a very strict abstraction: the reaction caused by an event is done in zero time. It is implemented by the popular languages Esterel and Lustre.
- The *timed model*, which introduces a so-called fixed logical execution time (FLET) regarding the invocation of tasks: Tasks are invoked within fixed periods and the output results are available only at the end of these periods. Giotto and TDL are two examples, which implement the timed model.

Kirsch worked out that only the timed model leads to predictable applications regarding time and value determinism. Applications programmed with the synchronous model behave only value deterministic, while the scheduled model fails for both criteria.

Wernicke [5] wrote in his paper about designing distributed applications with DaVinci that the runtime behavior of a DaVinci application depends on the prioritization of tasks and bus messages. Therefore, DaVinci applications are a classical example of the implementation of the scheduled model. As there is no explicit way to access common resources (e.g. sensors, communication channel), the usage of semaphores or OSEK resources may lead to phenomena such as priority inversion, where determinism cannot be guaranteed.

5.1.2 Composability

Another feature of the timed model is that software, which is written in this model, has a high reusability. This is because only the timed model fulfills the composability criteria, which means that Software components do not behave differently when they are combined with other components. DaVinci has no mechanism to ensure that. Especially if a set of components is reused and therefore combined with different components in a different environment, the behavior of the set of components and even of a single component is not the same.

5.1.3 Software Standardization

The behavior of a software component should be specified independently of its implementation. DaVinci tries to separate the platform specific components (like drivers) of an application from the platform independent components (called Software components). However, this separation does not include the timing behavior of the Software components. If the hardware or the operating system changes it is very likely that also the temporal behavior of the DaVinci application changes.

5.2 Use Case 1: Distributed Brake-By-Wire

5.2.1 Main Flow of Events

1) **Structural design with software components.** In this step, the software developer has to define the software components and their interfaces. In our case study, we use two software components: the first component controls the front-left wheel, while the second component controls the front-right wheel. Each component has three input parameters and two output parameters. Two of the input parameters are for reading the sensors, while the third input is used for reading information about the state of the other software component. The first output parameter is for controlling the brake, while the second output is given to the other component to inform it about internal states.

   Compared to the TDL development process, this matches exactly the definition of TDL tasks. Note, that the DaVinci software components are implemented as functions or methods, while TDL provides a module concept to build real software components, which should be a set of functions and methods.

2) **Implementation of the software components.** The implementation can be done either by describing a finite state machine (with tools like Simulink/Stateflow or Statemate Activity) or by specifying the corresponding C code functionality. Each software component calculates the control variable according to the sensor values of the wheel and the brake pedal. If the third input of the Software component indicates that there is something wrong with the other software component, a special algorithm is used to calculate the actuator value. In our use case, each software component is modeled in Simulink (presuming that the control law is known from the beginning).

   In TDL, the definition of the task functionality can be defined in Simulink or in C (or in any imperative language). Thus, this step is again very similar to the TDL development process. However, in TDL this step is already fully integrated into a simulation environment. As a result of the FLET assumption in TDL, machine independent and deterministic test cases can be formed up in a very early stage.

3) **Integration of the functionality.** The software components are connected to each other to form up the complete system, independent of any distribution aspects. In our use case, the software components are connected according the description of the case study. The sensors and actuators are connected to so-called DeviceAccessors, which are used to verify the logical behavior of the whole system.

In TDL, Simulink is fully integrated into development process. The equivalent to DaVinci's DeviceAccessors is a Simulink subsystem block, which defines the plant of the control system and is directly connected to the TDL controller model. The whole control system can be simulated in Simulink. A very important point at this stage is that the simulation is not only used to verify the logical behavior of the controller. The run-timing behavior of the task (and thus for the whole system) is usually already defined at this development step. The FLET assumption of TDL guarantees that the behavior of the system will be the same on any target platform. DaVinci does not provide anything that guarantees that at the same level.

4) *Integration of Software and Hardware.* In order to build a real system, the software components and their interactions have to be associated with a specific hardware and its topology. Concrete sensors and actuators have to be defined: DaVinci already provides a good deal of hardware specific drivers for the most common platforms (e.g. CAN Bus). The DaVinci SAR tool is used to define ECUs and each software component is mapped on a concrete ECU. Depending on the fact that interacting software components are placed on the same ECU or not, the communication is implemented via inter-process communication or via bus communication. DaVinci provides the necessary platform dependent infrastructure to implement this.

   In our use case, we define two ECUs to control the front wheel and the rear wheel, respectively. Each ECU is connected to two sensors (brake pedal and wheel sensor) and one actuator (wheel brake). On each ECU, one software component is placed. Since the two components exchange information about their states and are placed on different ECUs, communication is performed via a bus (normally CAN).

   TDL will provide a tool called platform editor, which allows defining such software/hardware mapping. In contrast to DaVinci, in TDL this hardware mapping is entirely independent of the defined controller regarding its run-time behavior.

5) **Parameterization of the runtime behavior.** This is a very important step in DaVinci as operating system tasks are created and the software components are distributed among these tasks. The runtime behavior of the system depends on the priorities associated with these tasks as well as on the priorities associated with messages, which are sent over the bus (see Wernicke [5]). In a real-time system, like our case study, the priorities have to be set in the way that every real-time constraint is met independently of the current system state. The drawback of this approach is that a system, which relies on prioritization for scheduling, is highly *non*-deterministic (see Kirsch [4]). In cannot be guaranteed that the runtime behavior of such a system is exactly the same if it is executed twice with the same inputs. In addition, if the software/hardware integration changes, the parameterization of the runtime behavior has to be adapted again.

   In TDL, the parameterization of the runtime behavior is already done after step 2) or 3). It is defined according the physical needs of the control system and is entirely platform independent. In addition, it is deterministic, no matter which platform is used and how the tasks are distributed over the system.

6) **Automated code generation.** In the final step, the code is generated using all information from the previous steps. If the software components are defined with modeling tools like Simulink, the code generation facilities of those tools are used. The DaVinci tool suite takes care about the right configuration of the bus driver, the interaction layer and the operating system (normally OSEK) as well as about the correct distribution of the software components. In our use case, code is generated for the two software components from a Simulink model. This code is compiled and linked against all configured device drivers provided by DaVinci for the OSEK operating system.

The compiled application is highly platform dependent since the runtime behavior is defined for a specific platform. In contrast, if the TDL toolchain is used so-called E-code is generated to handle the runtime behavior. E-code is platform independent and is interpreted by a virtual machine called E-machine. Only the E-machine has to be ported to a specific platform, which can be done very easily as a result of the small instruction set of E-code.

5.2.2 Analysis of the Application Runtime Behavior

Communication. Using the default communication system supported by DaVinci (CAN) there is no synchronization between the tasks running on both ECUs, and the system is completely unpredictable.

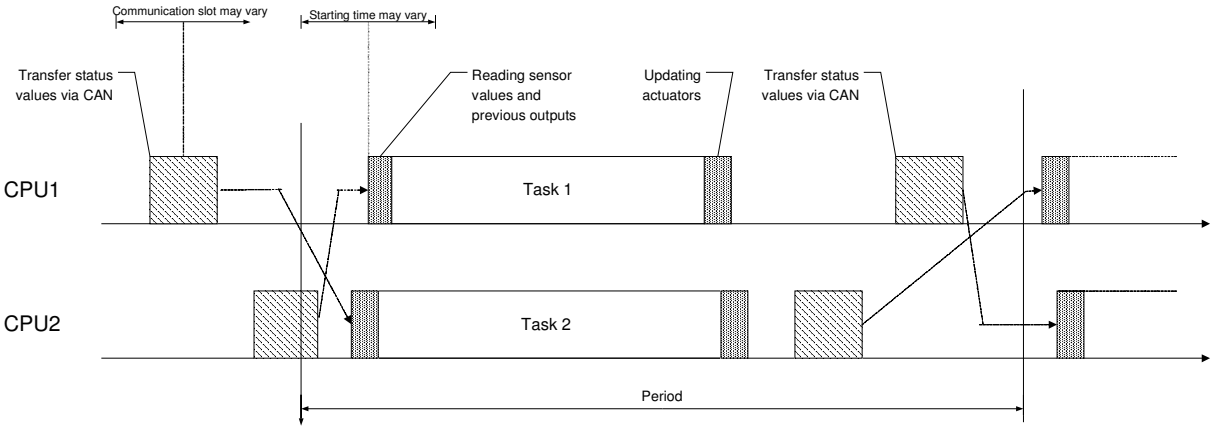

Figure 5.1 – Non-deterministic distributed system.

Time and value determinism. As each ECU runs only one task assuming that the tasks are run periodically based on a timer interrupt the system reads the input sensors at the right time, but the outputs will present jitter. This means that the values passed between ECUs might not be accurate and the whole system behaves in a non-deterministic fashion.

Using a time triggered communication bus may be a solution to synchronize the actions of both ECUs but the actuator updates of each ECU will still present jitter.

5.3 Use case 2 – ECU Consolidation

In this section we observe, which software development steps have to be repeated in DaVinci if the underlying platform changes significantly. In our case study, we consolidate the two ECUs to only one. In addition, the new ECU is a different and more

powerful ECU than the original ones. We step through the six software development steps of DaVinci and analyze the necessary action for each step in order to react to the platform changes:

5.3.1 Main Flow of Events

1) – 3) **Structural design with software components, implementation of the software components and integration of the functionality.** The software components of the new system should be the same as for the distributed system.

   Therefore, for DaVinci as well as for TDL (regarding task definition and task functionality) no actions are needed.

4) **Integration of Software and Hardware.** The new hardware architecture requires changes of the hardware/software mapping (including new drivers for sensors/actuators). Both software components have to be mapped to the same ECU. In DaVinci, a change in the hardware/software mapping leads to a different runtime behavior of the whole system because of there is no definition of the runtime behavior up to this software development step. The changes would be necessary even if the same ECU is used and another software component is simply added. Therefore, DaVinci fails regarding the criteria of compositionality.

   In contrast, TDL is composable because of changing the hardware/software mapping (e.g. adding a second task to an ECU) does not affect the runtime behavior of a TDL controller because of the fact that the behavior of each task is defined independently of the platform in a former development step.

5) **Parameterization of the runtime behavior.** In DaVinci, every change of the platform or the hardware/software mapping leads to a new parameterization of the runtime behavior. Currently, the DaVinci tool suite does not support porting existing applications to other platforms without changing their configurations.

   In TDL nothing has to be done here, because of the FLET assumption, which defines the TDL controller entirely platform independent.

6) **Automated code generation.** The generated code contains the functionality of both tasks but not their timing specifications. The system behaves differently on both platforms, even if we re-parameterize the runtime behavior for each platform.

   In TDL the E-code combined with the E-machine fulfills the criteria of software standardization. The E-machine ensures that the runtime behavior of the application is the same on every platform. In our use case, this means that it makes no difference if the application runs on two ECUs or consolidated on a single ECU.

5.3.2 Analysis of the Application Runtime Behavior

Determinism. The applications of both use cases, the distributed one as well as the reworked single CPU solution, expose non-deterministic behavior. The input values

will be read at various instants of time, depending on task priorities and their completion times (not WCET). The actuator output produces jitter because of the non-determinable start time and execution time of the two tasks. The dynamic scheduler of the OSEK OS utilized by the running system, further increase the non-determinism aspects of the application.

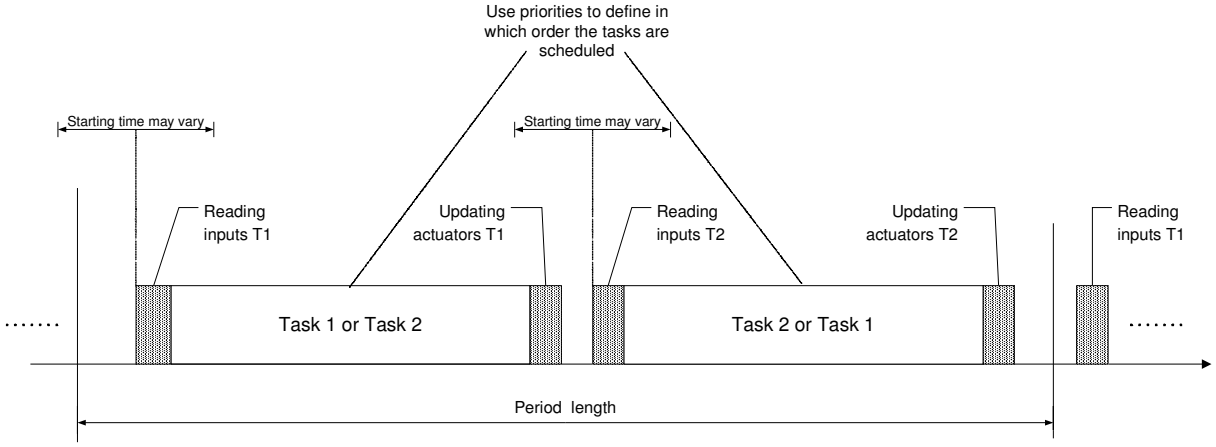

Figure 5.2 – Composed model of two tasks on one ECU.

Composability. The second use case shows, that composing different software components, even on only one CPU, leads to different run-time behavior of the application. Practically the result is a different application that has to be checked again and fully tested while still not being deterministic.

Software standardization. Changing the architecture of the system, the software has to be adapted to match the new hardware configuration. However, this does not imply the same behavior but requires complete rechecking of the application and more testing.

5.4 Summary

The DaVinci software development tools brings no new ideas in the area of designing real-time applications for embedded systems. It is only a software development environment for existing programming models (like the scheduled model). It has good bindings to existing modeling tools (such as Simulink) and integrates them into the development process. DaVinci provides tools to make the design of software for distributed systems easier, but does not provide anything that automates the development of software that is independent of the underlying platform and topology.

The two use cases show that the DaVinci tool suite does not support the development of applications that comply with the requirements of determinism, compositionality and software standardization.

6 dSPACE Tool Suite by dSPACE

6.1 Overview

In control engineering hardly any controller is designed without simulation. The typical procedure is to develop a model of the plant and simulate it on a computer. Then a controller is added to the simulation and the software is optimized.

dSPACE focuses on the following development procedure:

- Design of the controller and non real-time simulation (using Simulink): the functional behavior of the controller is determined, tested and revised.
- Testing of functional and temporal behavior on dedicated real-time hardware: the temporal behavior of the control algorithm can be measured and revised without having to supply the production target hardware for executing the control algorithm. The simulation hardware is connected to the real plant to feed the simulation with real world signals.
- Hardware in the loop (HIL) simulation: the model of the plant is simulated on the real-time simulation hardware enabling testing of the production hardware and software. Here erroneous behavior of the plant may be simulated and the behavior of the controller during test runs does not damage the real plant.

dSPACE supports this development process of embedded control applications with a complete tool chain from designing the application in Simulink over automatic C-code generation (via the code generator Targetlink) to real-time simulation of the application by executing code on simulation hardware. The dSPACE solution is also well suited for hardware in the loop (HIL) simulation, especially because of the powerful real-time hardware.

The “Control Desk” application manages simulation experiments either as Simulink simulation or as real-time simulation on dedicated dSPACE hardware. It handles the experiment management, downloads the code to the simulation hardware and interfaces, keeps track of all data generated by the model executed on the dSPACE hardware and allows changing the input parameters of the model in execution. The values that should be tracked during the experiment are assigned to the desktop of the Control Desk application interactively via drag and drop, and their display may be customized in various ways.

The behavior of the model can thus be determined and the model can be adapted and tuned. If the application has reached production state, the code generator may be used to generate production code for a specific target platform.

dSPACE supports distributed real-time simulation hardware: several CPU boards (up to 20) can be interconnected via Gigabit Ethernet to form a distributed (multi-processor) simulation system with globally synchronized time base. One of the boards acts as time master and sends a timing signal that is used as time source by the other boards. There is a significant delay regarding the transmission of the clock signal derived from the usage of the same Ethernet communication channel.

The achievable accuracy with a resynchronization period of 10ms is around 1µs. This globally synchronized time serves as time base to time-stamp generated output values. Thus, the values calculated for a distributed system on the different boards can be correlated according to their time stamps in order to obtain a consistent global view of the whole trace. Because of the globally synchronized time base, the temporal behavior of the whole system can be determined.

A model may be distributed among several hardware units with the help of dedicated Simulink blocks. Those blocks are used to model the transfer of signals between different computer boards by connecting a source hardware unit with a target hardware

unit via the communication channel (reflecting the communication over physical bus lines from the hardware level). If the underlying hardware topology or the distribution of the model changes, the changes also have to be incorporated into the Simulink model by changing the communication blocks. In the case of a production system where the communication is carried out over e.g. the CAN bus, the temporal behavior of the whole system cannot be predicted. It has to be simulated as whole.

6.1.1 Determinism

The dSPACE real-time kernel provides a priority-based preemptive scheduler that supports Rate Monotonic Scheduling. Depending on the system load, e.g. high interrupt frequency / high event load in the case of an alarm scenario, the starting time of a task may be delayed significantly, and in the worst case, its deadline may not be met. In addition, if shared resources are used within tasks, the priority based scheduling of tasks may lead to priority inversion problems and thus to non-deterministic temporal behavior of the system (refer to Kirsch [4]: problems of the scheduled programming model). Thus, the result of control calculations performed within those tasks may differ significantly, depending on when the computation of a task is actually carried out. If the periodic computation cannot be guaranteed to have minimal temporal jitter due to non-deterministic temporal system behavior, the actual values taken as input for the control computation would have to be latched periodically in a time-triggered manner in order to guarantee consistent behavior of the system.

Assume the following scenario: a control algorithm reads two real-time values to calculate its output. One value periodically computed by a remote node is transferred ‘immediately’ via the dSPACE Gigabit Ethernet link to the processing node and the second value is read directly from a local sensor within the computation task. Depending on the point in time the computation task is started, the value transferred from the remote node may already have been arrived a couple of milliseconds ago while the sensor value read locally is taken when the task is started. Thus, the age of the two different input values is not the same and the values are not comparable, that leads to inconsistent results depending on the difference of their age.

Explicit programming or modeling constructs would have to be introduced at Simulink level (e.g. delay blocks) in order to guarantee the temporal consistency of control computations designed with dSPACE/Simulink. In addition, programming guidelines regarding the usage of those blocks would have to be obeyed by the developers.

6.1.2 Composability

If the components of a dSPACE real-time target system are changed, e.g. a new component is added, the behavior of the whole system has to be tested whether it still matches the requirements. It cannot be relied on, that the behavior of the system, especially in the temporal domain, is still the same after the change. The reason is, that introducing new functionality usually causes increased load of the system resources. In a dSPACE single processor system the increased CPU load may result into different temporal scheduling behavior because of the priority based scheduling strategy. In a dSPACE multi-processor system the increased load of the communication channel may result in longer communication delays for some values. Thus, the whole system may behave differently after the introduction of a new component.

6.1.3 Software Standardization

The behavior of a control application intended for the dSPACE real-time hardware target platform can not be specified completely independent of the architecture of the target hardware. If for example the configuration of the target hardware changes, e.g. one CPU is removed, this change has to be dealt with in Simulink, representing the model / source code level. The blocks describing the connection between the system and the component that was removed would have to be modified. In addition, the blocks denoting the location of functionality on certain hardware units would have to be modified for that part of the functionality that has been residing on the removed CPU.

6.2 Use Case 1: Distributed Brake-By-Wire

6.2.1 Main Flow of Events

1) **Structural design in Simulink.** There are two functional blocks: T1 and T2 computing the control algorithms for each wheel. The dSPACE I/O blocks read the sensor values S1, P1 and P2 for computation task T1, respectively S2, P1 and P2 for task T2. The additional two dSPACE I/O blocks access the actuators.

   As the brake-by-wire application relies on the membership information of the participating nodes, it has to be guaranteed that this information is exchanged at predefined points in time. I.e., if the membership information of one of the nodes is not received by the others until a specific point in time it has to be assumed, that this node has failed. This has to be taken into account designing the inter process communication with the dSPACE IPC blocks: After each processing cycle, the actual values are transferred to the other node and vice versa (refer to Fig. 6.1). The successful reception of the message from a node also serves as membership information for that node.

2) **Implementation of functionality.** The functionality of the functional blocks T1 and T2 of the case study application is modeled in Simulink or can be hand-coded for such simple application.

3) **Platform specification.** In order to integrate the application model with the available distributed hardware platform you must setup the options for building the multiprocessor model in dSPACE Control Desk. The topology information of the multiprocessor system must be specified prior to the first build of the model. This has to be done only once for each new application / hardware configuration combination.

4) **Integration of functionality.** The model is distributed using dedicated dSPACE blocks in Simulink, i.e. by modifications on model level (Refer to dSPACE [6]):
   a) The whole model is defined as multiprocessor model using the dSPACE multiprocessor setup block.
   b) The two functionality blocks (T1 and T2 of the case study) are assigned to dSPACE timer tasks. Note: If a model part is not explicitly defined as driven by an interrupt, it is part of a timer task by default. The timer tasks have to be invoked cyclically with the same frequency at both CPUs: See step 5)

c) The model is distributed using the dSPACE IPC blocks introducing communication between the two functionality blocks according to the application specifications. The IPC blocks implicitly define the assignment of the functionality blocks to the different CPUs. The behavior of the IPC is set to *swinging buffer, unsynchronized* (refer to dSPACE [6], p 123) with the help of the *protocol* property of the block parameters dialog of the IPC blocks.

5) **Parameterization of the runtime behavior.** To determine the timing behavior of the dSPACE application, the period of the computation cycle has to be set. This is done within the property dialog of timer task block. If there are multiple applications on the same CPU and the system is thus run in *multiple timer task mode*, the priority of the tasks has to be set within the *RTI Task Configuration Dialog* accordingly.

6) **Automatic code generation.** The model for the distributed platform is compiled by pressing the *Build All* button within the *Multiprocessor Setup* dialog. *Real-Time Workshop* automatically generates the functionality code for all CPUs. In order to transfer the executable code to the dSPACE real-time hardware you have to press the *Download* button.

6.2.2 Analysis of the Application Runtime Behavior

Communication. The swinging buffer protocol of the dSPACE IPC unsynchronized communication is used for communication. The most important property of this protocol is that the reader of the message is not blocked until the message has been arrived. Within a tightly coupled model it would cause a delay of one cycle, because the consumer always reads the value of the last computation meanwhile the producer is computing the next value during the actual cycle. For the brake-by-wire application it is appropriate, because each node receives at the begin of each cycle the status information of the other node of the last cycle and both nodes can compute the values of the actual cycle in parallel without having to wait for the values of the other node. Otherwise, the computation times of both nodes would sum up instead of being able to compute in parallel.

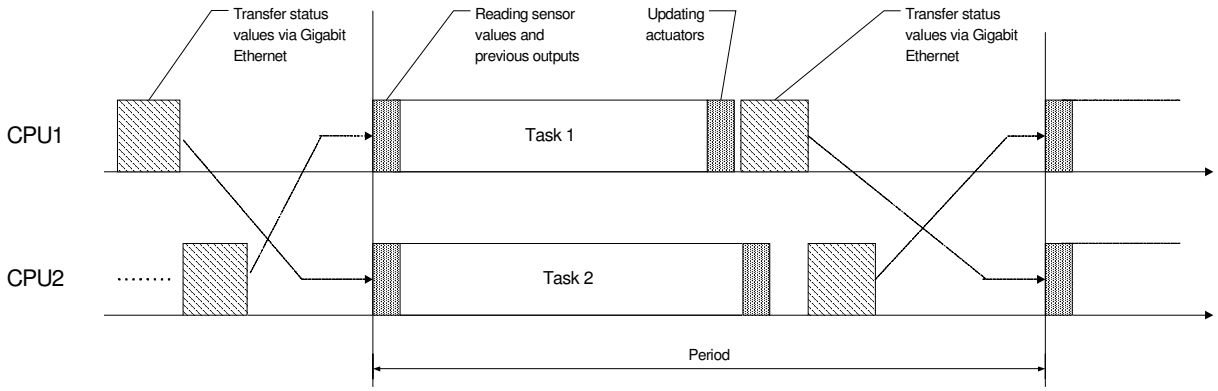

Figure 6.1 – Runtime behavior of the distributed brake-by-wire model.

Temporal behavior. The sum of the computation time of the task (WCET) and the sum of the communication time of the messages of both nodes must not exceed the length of the period. Otherwise, old values of the cycle before the last cycle are read at the beginning of each cycle. As long as the brake-by-wire application task is the only

task running on the CPU, the application behaves deterministic regarding the time when the input values are read and the values passed between the two components, because those values are also accessed only at the beginning of each period (refer to Fig. 6.1).

Value determinism. Depending on the computation time of the tasks, the update of the actuators may happen at different points in time within the period (output jittering) and thus lead to different devolution of the brake-force at the different wheel nodes. It does not behave deterministic regarding the output values. In the future, when X-by-Wire applications will become industrial praxis, it is also very likely that functional parts of other steer/brake assistance applications run on the same wheel node CPUs, like e.g., anti lock braking system, electronic stability program, brake assistance and adaptive speed control. Because of the priority based dynamic scheduling of those concurrently running application tasks, the point in time when the brake-by-wire task is actually scheduled within its period could be postponed to an arbitrary point in time (see Fig. 6.2). In this case, the value determinism is completely lost. The input values taken to calculate the control laws may differ from one period to the next and the computed results can not be reproduced.

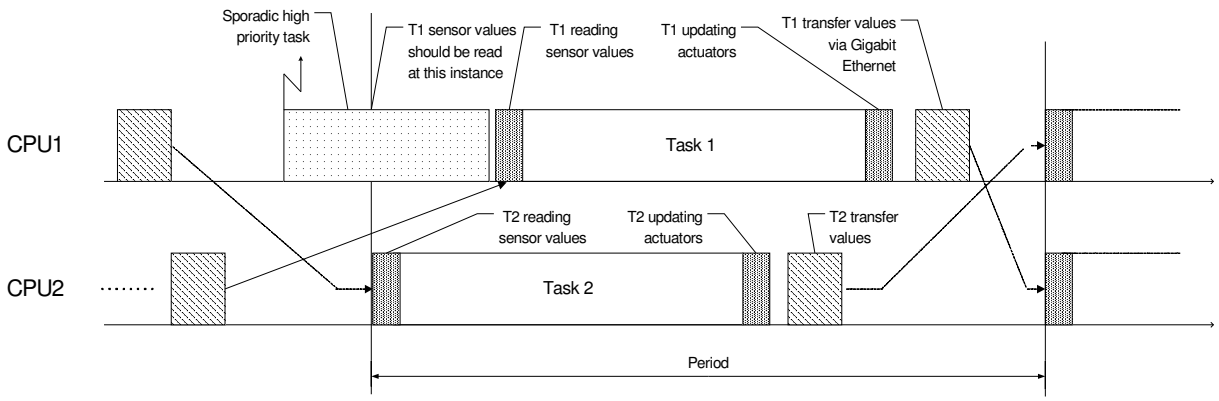

Figure 6.2 – Other applications running in parallel on the same CPU.

6.3 Use Case 2: ECU Consolidation

6.3.1 Main Flow of Events

1) **Structural design in Simulink.** According to the requirements of the second case study, the structural design of the application has to be preserved. However, this is not possible with the dSPACE development environment, as converting the distributed Simulink model into a single processor model implies reworking the model in order to obtain equivalent deterministic behavior as the distributed solution. Replacing the IPC blocks with direct links is not sufficient, because the model behaves completely different now: the two functional blocks T1 and T2 run in sequence on a single CPU no matter their priorities, because there is no master controller to preempt them and switch control from one to the other (ex. in EDF or RM scheduling). Thus the two functional blocks read their inputs and update their outputs at different points in time (see Fig. 6.3), instead of at the same point in time (as in the distributed solution), which is likely to result in a completely different behavior of the application.

2) **Implementation of functionality.** According to the requirements of the second case study, the implementation of functionality of the application has to be preserved. The functional blocks T1 and T2 will be the same in this case.

3) **Platform specification.** In order to adapt the model to the new situation you have to change the options for the model building process in the *simulation parameter* of the Control Desk dialog. The target CPU has to be fast enough to run each task at least twice as fast as on the CPUs of the distributed solution, in order to be able to compute the control algorithm within the same time as the distributed hardware and thus to guarantee the same controlling behavior of the application.

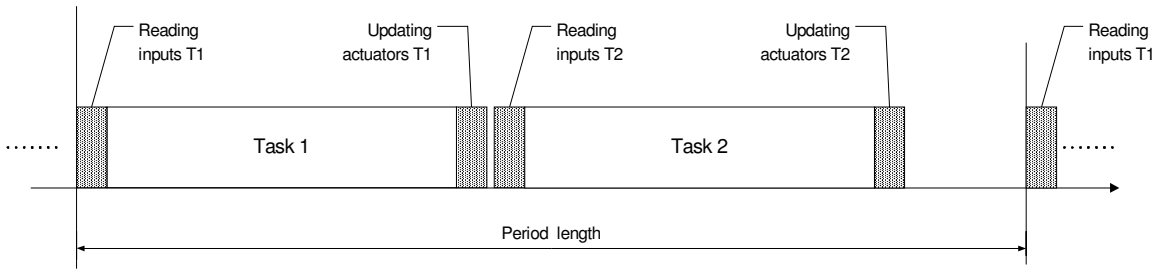

Figure 6.3 – Behavior of the consolidated brake-by-wire model.

4) **Integration of functionality.** The consolidation step requires the modification of the Simulink model:
   a) Remove the dSPACE IPC blocks between the two tasks T1 and T2. Add direct connections for the values passed between the two components. This will make the multiprocessor model a single processor model.
   b) Add unit delay blocks to determine the flow of the consolidated model and to avoid algebraic loops: E.g. functional block T1 running before functional block T2. However, this single CPU model will not even approximately behave like the distributed solution. Thus, the single CPU model obtained by replacing the IPC blocks has to be reworked in order to obtain deterministic behavior, which is equivalent to the behavior of the distributed model.
      Remark: The more complicated the distributed model is the more effort has to be spent to modify the model in order to make the single CPU solution to behave roughly like the distributed solution.
   c) Status values (membership, revolutions of the controlled wheel) from T1 to T2 and vice versa must be passed in a different way. T1 may directly read the status values of T2 of the last cycle, but T2 must not read the values produced by T1 in the current cycle but it has to read the values produced by T2 during the last cycle. Thus the values produced by T1 during the last cycle have to be buffered (refer to Fig. 6.4).
   d) The sensor values of both wheels have to be read at the begin of the period, the actuator values have also to be written for both wheels at the same point in time to keep deterministic behavior of the application. The sensor reading of T2 has to be moved to the begin of T1, the actuator update of T1 has to be moved to the end of T2. (see Fig. 6.4).
   e) The sensor and actuator drivers may have to be adapted to the new hardware architecture.

5) **Parameterization of the runtime behavior.** The period of the computation cycle of T1 and T2 has to be set within the property dialog of the timer task block T1 and T2 are assigned to. The frequency is set to the same cycle length as the application cycle length of the distributed solution (see Fig. 6.3 – parameter *Period length*). If there are additional applications running on the same CPU, the priority of the tasks has to be set within the *RTI Task Configuration dialog* accordingly. Note that in a multi-application set-up the runtime behavior of the whole system may be non-deterministic (see below the runtime behavior analysis).

6) **Automatic code generation.** Building the model for the single ECU platform: select the *Tools – Real-Time Workshop – Build Model command* from the menu bar of the model. *Real-Time Workshop* automatically generates the code. In order to transfer the executable code to the dSPACE real-time hardware you have to press the *Download* button.

6.3.2 Analysis of the Application Runtime Behavior

The reworked model will still not behave timing/value deterministic, because the time of the actuator updates can not be determined in advance. Jittering on output ports is increased by this solution as the actuator updates depend on the sum of the execution times of T1 and T2.

As with the distributed model, if there are other applications running in parallel on the same CPU, the priority based scheduling of the dSPACE run time system will cause non-deterministic behavior of the brake-by-wire application, because the points in time when inputs are read and outputs are updated cannot be determined.

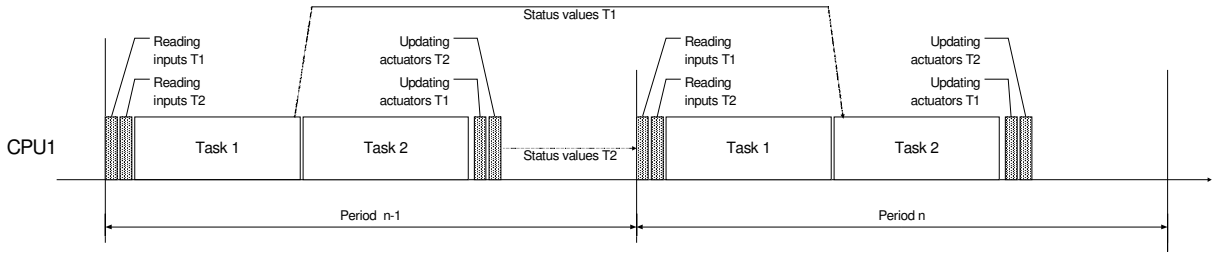

Figure 6.4 – Behavior of the reworked consolidated brake-by-wire model.

Determinism. The applications of both use cases, the distributed one as well as the reworked single CPU solution, expose non-deterministic behavior. The actuator output produces jitter because of the non-determinable execution time of the two tasks. If there are additional applications running on the same CPU, determinism is also lost for the input values.

Composability. The second use case shows, that composing different software components, even on only one CPU, leads to different runtime behavior of the application.

Software standardization. Changing the architecture of the system, the software has to be adapted to match the new hardware configuration. Use case two shows that the source code, i.e. the Simulink model has to be reworked in order to maintain similar behavior of the application.

6.4 Summary

The dSPACE development environment supported by the powerful dSPACE real-time hardware focuses on rapid control prototyping: developing and simulating the functional behavior of a control algorithm in Simulink, testing and optimizing the behavior on the real-time hardware as well as HIL simulation of the productivity controller. This approach clearly focuses on the target platform and is thus in line with the "programming to the platform" strategy which is very common in the control application development community. The drawbacks of this approach: major postulates of software engineering are bypassed and the developed code is not standardized and not composable. Moreover, the control application does not behave deterministically if no special measures are taken.

The two use cases show that the dSPACE development environment and the dSPACE hardware fail in supporting the development of applications that comply with the requirements of determinism, compositionality and software standardization.

7 TDL Tools by University of Salzburg

7.1 Overview

The TDL tool chain consists of a TDL compiler, which translates a TDL module into executable E-code and an E-machine, which executes the E-code. In addition there may be platform specific files produced by the compiler and used by the E-machine. The TDL compiler provides a plug-in interface for supporting arbitrary target platforms. The E-machine may also be combined with the E-code in an executable image if the platform requires statically defined binary images for execution. For distribution, TDL provides a bus scheduling tool, which takes a bus description and node list as input and produces a bus schedule as output, which will be used by the E-machine as additional input. For integration with Simulink a translator is provided. The bus scheduler and the Simulink translator are work in progress.

7.1.1 Determinism

TDL is based on the FLET assumption where each task has precise logical starting and ending time, regardless of when it is actually scheduled to run and finish within its period. This assumption abstracts from any specific scheduling strategy or real-time OS and provides timing and value deterministic results with minimum jittering (close to zero) on the output ports. The sensors are always read at the beginning of the logical starting time of a task and the outputs of the tasks are available only at the end of a task's logical execution time. The behavior of a TDL program is solely determined by its physical environment and not by CPU performance, bus load or scheduling strategy.

7.1.2 Composability

TDL has been designed around the notion of a module as the building blocks of applications. Each module provides a namespace for constants, types, sensors, actuators, tasks, and one or more modes that define a specific state of the application. A mode may contain one or more task invocations that run logically in parallel with possibly different periods, actuator updates and conditional mode switches. Data exported from one module can be imported from another module, or multiple modules

can be imported to statically define the components of an application. Composing an application of multiple modules means that all modules are executed in parallel.

Besides functional composition, TDL modules also provide temporal composability. because the timing behavior is based on fixed logical execution time (FLET). This introduces the freedom to schedule task execution such that the logical timing behavior of all executing modules is preserved and also allows for distribution of modules between nodes in a networked system without changing the timing.

7.1.3 Software Standardization

TDL is a high level language for specifying real-time applications. It separates the application model from functionality code, it abstracts from the execution platform and from the network topology. This separation allows maintaining the behavior of the whole system even if the underlying hardware platform, operating system or network topology is changed. What is needed is an implementation of the E-machine for a particular platform and, depending on the target platform, a TDL compiler plug-in which generates additional target specific files.

7.2 Use Case 1: Distributed Brake-By-Wire

7.2.1 Main Flow of Events

1) **Structural design of the application.** In TDL the structure of the application is defined by means of TDL software modules and their import relations. The architecture of the case study application consists of two functional blocks: T1 and T2, each computing the control algorithm for one wheel. The TDL model for this case study consists of two modules M1 and M2, each containing one task which executes the functionality code Task 1 respectively Task 2. Each of these two modules has two sensors for the pedal, one input port to read the status information produced by the other module, one output port to pass its status information to the other module and one actuator output to control the brake actuator. The structure of the application is either defined as TDL program in textual form or is modeled in Simulink together with the graphical TDL editor.

   The import relation between M1 and M2 in this particular example forms a cycle because M1 imports M2 to get access to the status information of M2 and vice versa. Since this is (at least currently) not allowed in TDL, we need to split the modules in order to break the cycle. The resulting module import diagram is shown in Fig. 7.1, where the modules have been given meaningful names.

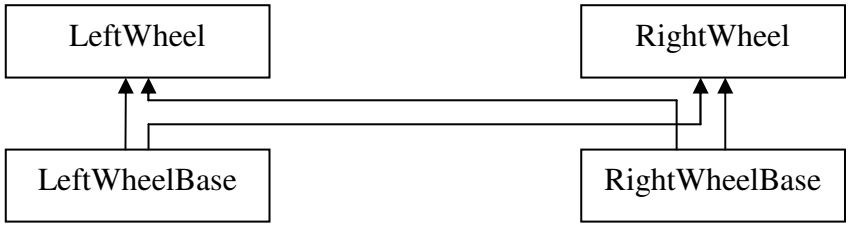

Fig. 7.1 Module import relationship without cycle.

The following code shows the TDL model for the left wheel. The right side would be denoted correspondingly.

```
module LeftWheelBase {

   public sensor
      int rpmLeft uses getRpmLeft;
      int p1 uses getP1;
      int p2 uses getP2;

   public task liveTask [wcet=2ms] {
      input int p1; int p2; int rpm;
      output int status := 0;
      uses liveTaskImpl(p1, p2, rpm, status);
   }

   start mode main [period=20ms] {
      task [freq=1] liveTask(p1, p2, rpmLeft);
   }
}

module LeftWheel {
   import LeftWheelBase as LWB;
   import RightWheelBase as RWB;

   actuator int brake uses setBrake;

   public task brakeTask [wcet=5ms] {
      input int p1; int p2; int rpm; int rightStatus;
      output int force := 0;
      uses brakeImpl(p1, p2, rpm, rightStatus, force);
   }

   start mode main [period=20ms] {
      task [freq=1]
         brakeTask(LWB.p1, LWB.p2, LWB.rpmLeft,
                                   RWB.liveTask.status);
      actuator [freq=1] brake := brakeTask.force;
   }
}
```

The two TDL modules LeftWheelBase and LeftWheel form a unit containing the control functionality for the left wheel, RightWheelBase and RightWheel form a unit containing the control functionality for the right wheel. They correspond to the functional blocks T1 and T2 of the case study. The two wheel units are interconnected via the liveTask.status output ports (refer to code fragment above) used to exchange information between the two wheel units.

2) **Implementation of functionality.** The functionality of the functional blocks (brakeImpl) is either modeled in Simulink or hand-coded (e.g. in C language). If the application is modeled in Simulink, usually also the software structure and the timing related aspects of the application (TDL code) is modeled in Simulink. The Simulink/TDL Translator automatically translates the TDL related part of the model (software structure and timing) into TDL code while an embedded code generator produces the functionality code out of the functional part of the model.

The sensor inputs rpmLeft, rpmRight, P1 and P2 are connected to the sensor implementation functions by the according *sensor* statements in the TDL program. The actuator outputs *brake* are connected to the brake actuator functions by the according *actuator* statements.

3) **Generation of E code.** After steps 1) and 2) have been completed E-code can already be generated for each of the TDL modules. Depending on the target platform also platform dependent code (e.g. OSEK OIL files) are generated.

4) **Mapping the model to the platform.** The software components have to be mapped onto a specific hardware platform. Using TDL this is done by specifying the *node assignment file*. This file can be generated graphically using the *TDL Platform Editor*. The file contains information about which TDL module is assigned to which ECU within the system. The TDL program itself contains no platform dependent information at all.

   Remark: The communication requirements are inferred automatically by the TDL compiler from the import relationships and the assignments of modules to nodes. The necessary messages are automatically generated and scheduled by the TDL Software Bus Compiler.

5) **Compilation of the System.** From the E-code generated in 3) and the *node assignment file* defined in 4) the TDL Bus scheduler generates a platform specific *tool configuration file* and the runtime part to interface the communication system. The platform specific *tool configuration file* is the input for vendor specific bus scheduling tools which set up the communication subsystem and specific RTOS parameters. The runtime part consists of appropriate driver calls to access the messages at the communication subsystem at particular time.

7.2.2 Analysis of the Application Runtime Behavior

The distributed TDL application runs strictly time-triggered. The TDL runtime (E-machine) are synchronized via the communication subsystem (which has to support this feature). Thus it can be guaranteed that sensors and actuators are accessed at particular predefined times and that those times are synchronized between the different ECUs. Figure 7.2 depicts the timing of the TDL application of case study 1.

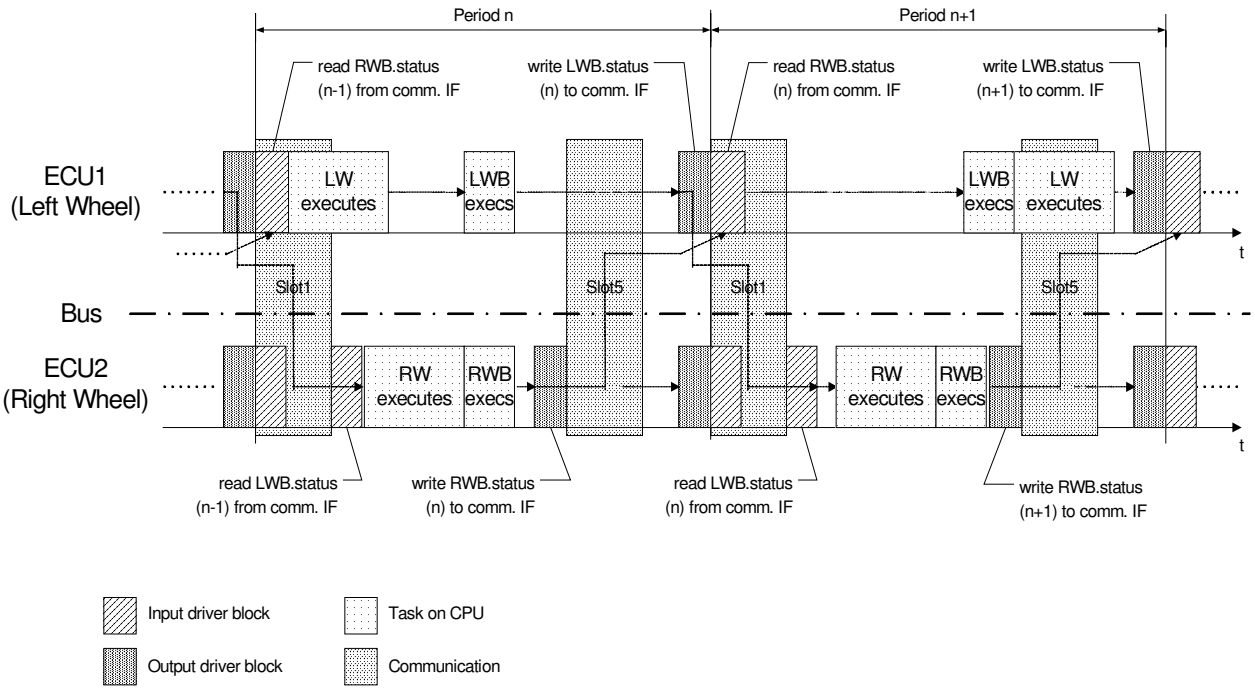

Figure 7.2 – Timing of the distributed TDL application.

At the beginning of each period the input drivers are called. They fetch the input values of the sensors (rpmLeft, rpmRight, P1 and P2) and pass them for usage to the tasks LeftWheel (LW), LeftWheelBase (LWB), RightWheel (RW) and RightWheelBase (RWB). At the end of each period the output drivers are called, they write the actuator values to the brake actuators. The communication of the status values LeftWheelBase.LiveTask.status (LWB.status) and RightWheelBase.LiveTask. status (RWB.status) happens during some communications slots (like 1 and 5 in this example) of the time-triggered communication subsystem.

Remark: this is only one out of many possible examples how the schedule of the communication slots can be done. During slot 1 LWB.status is transferred from ECU1 to ECU2 and is available at ECU2 at the end of slot 1. Thus task RW which consumes this value must not be scheduled before the end of slot 1. The additional input driver block on ECU2 at the end of slot 1 fetches the value LWB.status from the communication controller and passes it to the task RW. The status value RWB.status is transferred during slot 5 from ECU2 to ECU1. There is an additional driver call before the beginning of slot 5 which passes the TDL variable RWB.status to the communication controller of ECU2. Thus RWB has to be finished before the beginning of slot 5. Because in this example there are only scheduling restrictions for RW and RWB, the driver calls accessing RWB.status respectively LWB.status at the communication controller at ECU1 happen within the driver blocks at the beginning respectively at the end of the period of LW / LWB.

As depicted in Fig. 7.1, every period the status values produced by LWB at ECU1 during the last period are available to RW at ECU2 and vice versa. Also sensors and actuators are always accessed exactly at period intervals.

7.3 Use Case 2: ECU Consolidation

7.3.1 Main Flow of Events

1) **Integration of Software and Hardware.** Because of the target hardware platform was changed for case study 2 (single CPU platform instead of distributed hardware), the *node assignment file* has to be adapted to reflect the changed assignment of software modules to hardware units (refer to Fig. 3.4). All TDL modules for both wheel units are now assigned to the single CPU ECU1 using the *TDL Platform Editor*. Because of the hardware changes it may be required to change the getter and setter implementations.

2) **Generation of E-code.** Using the unchanged source code of case study 1 E-code and platform dependent code for the new target platform is generated.

7.3.2 Analysis of the Application Runtime Behavior

The temporal behavior of the single CPU TDL application is strict in line with TDL semantics: Tasks LW and LWB respectively tasks RW and RWB are executed every period. Sensors and input ports are read at the beginning of each period, actuators and output ports are written at the end of each period as illustrated in Fig. 7.3. Synchronization between the two TDL modules happens implicitly due to execution on the same E-machine. The status values are exchanged by LW accessing the output port RWB.status and RW accessing the output port LWB.status.

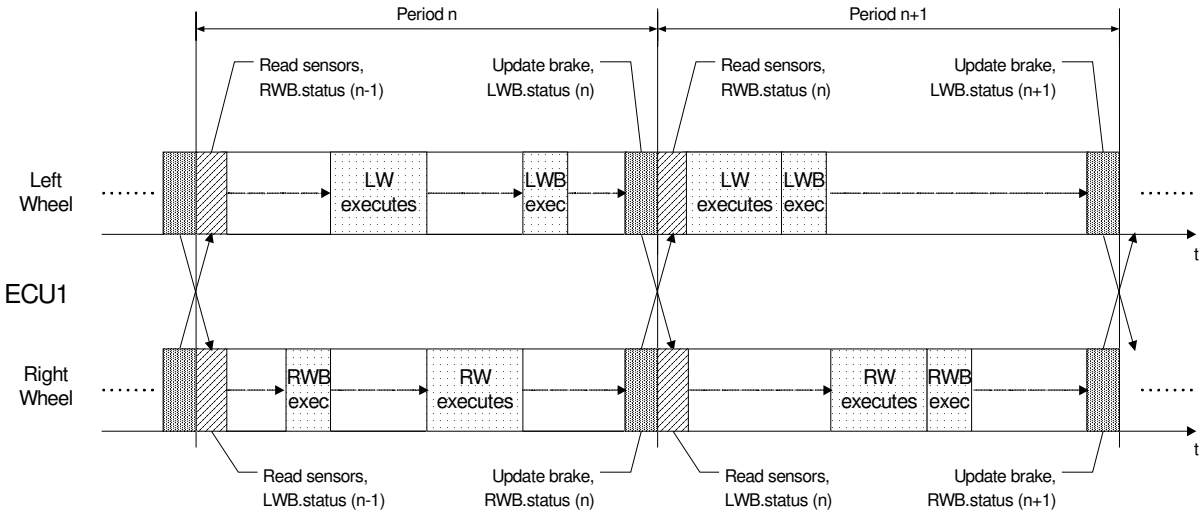

Figure 7.3 – Timing of the consolidated TDL application.

Because of that fact, the temporal behavior of the consolidated TDL application of case study 2 regarding input and output values is exactly the same as that of the distributed TDL application of case study 1. It thus exposes exactly the same temporal behavior as the application of case study 1.

Determinism. The TDL applications of both case studies show the same deterministic behavior: Input and Output values are accessed at precisely determined points in time. The behavior of the application is predictable and can be reproduced. Moreover, the jitter on the output ports is minimum.

Composability. The second case study shows that an existing TDL application could be ported to a different hardware platform still exposing the same temporal behavior as on the original platform.

Software standardization. Changing the hardware architecture of a system, an existing TDL application can easily be ported to the new platform without the need of adapting a single line of source code. Only the platform annotation file has to be adapted in order to reflect the assignment of TDL software modules to hardware units.

Summary

The TDL programming paradigm enables the development of embedded real-time applications that fulfill the requirements of determinism, composability and software standardization. Determinism is a must for real-time software anyway, especially if the software is targeted for safety-critical applications. It is the basis to proof that within the fault-hypothesis the behavior of the system will always be predictable. But also the requirements of composability and software standardization will become more and more vital for embedded applications in the future as the degree of distribution and thus the complexity of embedded real-time systems will grow.

8 Comparison Summary

Although it is a difficult task to compare commercial products to a fairly new academic approach, the comparison indicates that the available products fail to achieve the goals aimed at by Giotto and TDL. The following table summarizes what we found in the course of the comparison.

Requirements		**dSPACE**	**TTTech**	**DaVinci**	**TDL**
Determinism	Time	No (1)	Yes (2)	No	Yes
	Value	No	Yes	No	Yes
	Jitter Free	No (3)	Yes (4)	No	Yes
Composability	Functionality	No	No	Yes	Yes
	Timing	No	No (5)	No	Yes
Software standardization	Platform changes (6)	No	No	No	Yes
	Topology changes (7)	No	Yes	No	Yes

Notes:
(1) By default the behavior is not deterministic. However in special cases an application may be designed to behave deterministically.
(2) By default the behavior is deterministic because of the time-triggered execution of tasks. However, regarding output values it may not behave deterministic.
(3) The execution time of tasks influences the output timing.
(4) It behaves jitter free on the bus level however the task execution times may introduce jitter at the outputs (direct) of the nodes.
(5) Adding a component implies adding more messages to the bus schedule.
(6) Changes of the hardware and/or operating system.
(7) Changing the number of the nodes in order to distribute or to consolidate an application.

TDL allows a platform-independent specification of the timing and commmunication behavior of a control system. This implies significantly improved reusability and

portability of TDL applications compared to state-of-the-art approaches considered in this study.

In addition to that, TDL has introduced the module construct as basis for component-based, modular development of embedded control systems. None of the other systems and tools offers a comparable abstraction.

Finally, TDL components can easily be distributed without taking the distributed platform into account from the beginning. In other words, TDL abstracts from the distributed platform. The platform details are specified separately from the TDL program and later in the development cycle.

9 References

[1] J. Templ: TDL Specification and Report. *http://www.softwareresearch.net/site/publications/C055.pdf*.

[2] Henzinger, T., Horowitz, B., Kirsch, Ch.: Giotto: *A Time-Triggered Language for Embedded Programming*. Proceedings of the IEEE, Vol. 91, No. 1, January 2003.

[3] American National Standard for Telecommunications: *Telecom Glossary 2000*

[4] C. Kirsch: *Principles of Real-Time Programming*, EMSOFT 2002

[5] M. Wernicke: *New Design Methodology from Vector simplifies the Development of Distributed Systems*, Vector Informatik Press Release, June 2003

[6] dSPACE GmbH: *Real-Time Interface (RTI and RTI-MP) Implementation Guide*, Manual v4.0, August 2003

[7] dSPACE GmbH: *DS1005 PPC Board Features*, Manual v4.0, August 2003

[8] dSPACE GmbH: *DS1005 PPC Board RTLib Reference*, Manual v4.0, August 2003